Technical University of Berlin

Project Report

Industrial Distributed Control Systems

Development of Deep Reinforcement Learning algorithm for Autonomous Robot Navigation

Winter Term 2019/20

Georg Kruse
Pavan Chitradurga Thammanna
Shringar Sham Rao

Department: Industry Grade Networks and Clouds
Professor: Prof. Dr.-Ing. Jens Lambrecht
Supervised by: M.Sc. Linh Kästner

Abstract

This project has been carried out as part of the Industrial Distributed Control Systems project module under the Department of Industry Grade Networks and Clouds in the Faculty for Electrical Engineering and Computer Science (IV) at the Technical University of Berlin. This project attempts to demonstrate the feasibility of using a Deep Reinforcement Learning algorithm such as Deep Q - Networks (DQN) for Autonomous Navigation using the latest Open Source Tech-stack available such as ROS2, gazebo9, and turtlebot3 Machine Learning packages (Keras and Tensorflow). It lays out the various processes that were followed to install, setup the code, and discusses any limitations faced in the process. The packages used are studied for their scalability and flexibility to tweak the various parameters and functions, and thereby enhance the learning capabilities of the model. The training process was conducted on the DQN model in two different simulation environments, solely relying on Laser Distance Sensor data for navigation. The results recorded show successful navigation attempts in a relatively low number of episodes and includes several observations regarding how the various parameters chosen, affect the training outcomes. The scope of modifying these values and extended training and simulation capabilities have been briefly explored in the report.

Contents

1 Introduction

Learning, by being acutely aware of how an environment responds to actions performed and the need to influence or control it, forms the foundational idea underlying nearly all theories of intelligence. This concept that molds the basis of intelligence in humans, although may seem to be too broad to grasp in its entirety, has created the means for developing Artificial Intelligence in specific fields where the application is bound by a domain. Moreover, if the possible interactions within an environment are relatively deterministic, then such theories of learning can be directly applied in computational problems where human-like decision making becomes crucial. [1]

However, the kinds of learning algorithms required for various classes of computational problems are dependent on one main factor. As imaginable, the fact that they have a high computational cost is a given, since such problems are usually demanding. However, whether or not they require vast amounts of data, including labeled data, solely depends on the kind of computational problem. Thus the scale and how the data is available forms this main factor.

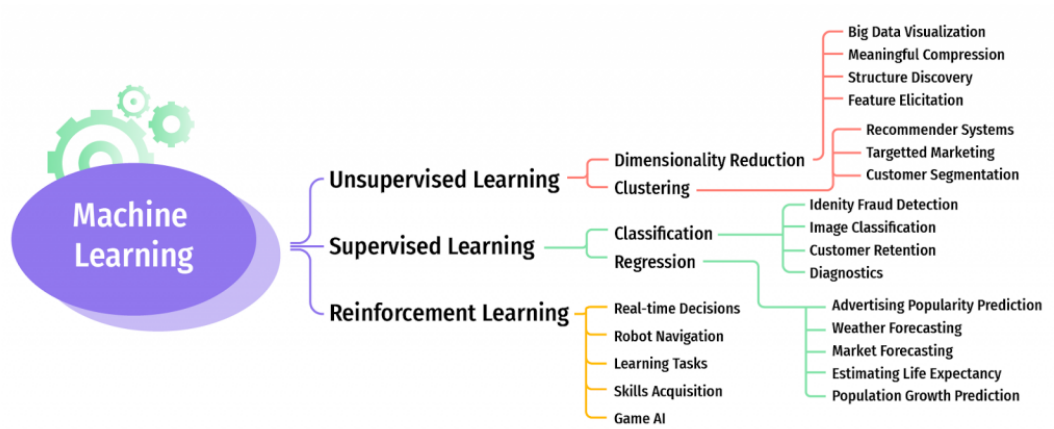

Figure 1.1: Types of Machine Learning and their Applications [2]

There are three main types of Machine Learning. Unsupervised Learning is usually applied in cases where data is available, but may not be labeled. Here, various algorithms are applied to identify patterns or structures that exist within the data. Supervised Learning is applied usually when sufficient labeled data exists and the model can be trained to identify relationships between a training set of data and the labels. The

trained model is then used to predict or generalize the labels for new data, based on the observations made in the training set. However, for a real-time scenario with a specific environment, the situation needs to be assessed and a related action needs to be taken. Here, the data required also relies on the ability to access information about the environment dynamically. This is where Reinforcement Learning comes into the picture.

1.1 Reinforcement Learning

Reinforcement learning is a general framework where agents (usually representative of simulation objects, characters in a game or robots), learn to perform actions in an environment to maximize a reward. The two main components are the environment, which represents the problem to be solved, and the agent, which represents the learning algorithm. The agent and environment continuously interact with each other. At each time step, the agent takes an action on the environment based on its policy $\pi(a_t|s_t)$, where s_t is the current observation from the environment, and receives a reward r_{t+1} and the next observation s_{t+1} from the environment. The goal is to improve the policy to maximize the sum of rewards (return). [3]

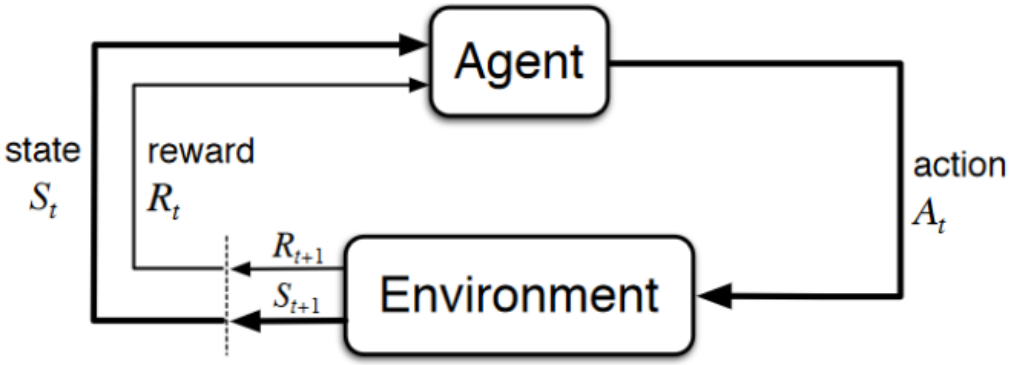

Figure 1.2: Reinforcement Learning [4]

This figure only shows a general framework along which many other sub-types of Reinforcement Learning algorithms have been modeled. The choice of the type of algorithm that needs to be used is dependent on the application. This general model can be extended to include the capabilities of the agent and the changes that could occur dynamically in the Environment, as well as the computational capabilities of Machine Learning algorithms that already exist, to best, assess the data obtained.

1.2 Markov Decision Process

The Reinforcement Algorithm mainly relies on the agent's ability to make decisions based on the state. Each state within an environment is a consequence of its previous state which in turn is a result of its previous state and so on. However, storing all this information, even for environments with short episodes (where an episode is a sequence between the first state and a terminal state) is infeasible.

To resolve this, the Markov property is applied to each state, i.e., each state depends solely on the previous state and the transition from that state to the current state. [5]

Mathematically, a state s_t satisfies the Markov property, if and only if,

$$P\,[\,S_{t+1} \mid S_t\,] = P\,[\,S_{t+1} \mid S_1, ..., S_t\,],$$

the state captures all relevant information from history.

For a Markov state S and successor state S, the state transition probability function is defined by,

$$P_{ss'} = P\,[\,S_{t+1} = s' \mid S_t = s\,]$$

It's a probability distribution over next possible successor states, given the current state.

A Markov process is a memory-less random process, i.e. a sequence of random states S1, S2, ... where each state has the Markov property. A Markov process or Markov chain is a tuple (S, P) on state space S and transition function P.

Similarly, A Markov Reward Process or an MRP is a Markov process with value judgment, saying how much reward accumulated through some particular sequence that we sampled. An MRP is a tuple (S, P, R, γ) where S is a finite state space, P is the state transition probability function, R is a reward function where,

$$R_s = E\,[\,R_{t+1} \mid S_t = s\,],$$

it says how much immediate reward we expect to get from state S at the moment. [6]

The agent tries to get the most expected sum of rewards from every state it lands in. In order to achieve that we must try to get the optimal value function, i.e. the maximum sum of cumulative rewards. Using Bellman equation, the value function will be decomposed into two part; an immediate reward, R_{t+1}, and discounted value of the successor state $\gamma(S_{t+1})$, where γ = [0,1] is the discount factor [0,1], giving us the following Bellman's equation for MRPs,

$$v(s) = E\,[\,R_{t+1} + \gamma\, v(S_{t+1}) |\, S_t = s\,] \text{ [6]}$$

1.3 Q Learning

Q-learning is a model-free reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward overall successive steps, starting from the current state. The Q-learning algorithm can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. [7]

The algorithm achieves this by performing a sequence of actions that will eventually generate the maximum total reward (i.e. the policy stated above). Here, the total reward is called the Q-value and is calculated as follows:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The above equation states that the Q-value yielded from being at state s and performing action a is the immediate reward r(s, a) plus the highest Q-value possible from the next state s'. Gamma here is the discount factor that controls the contribution of rewards further in the future. The Q-value is also dependent on the subsequent states and can be calculated as follows:

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) \dots\dots\dots \gamma^n Q(s''^{\dots n},a)$$

This is a recursive equation that can have arbitrary values. However, the best way to converge these values into an optimal policy has been shown by including a learning rate as follows:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\Big]$$

Here, α is the learning rate and determines exactly to what extent newly obtained information overrides the information that already exists. [5]

1.4 Deep Q-Networks

Q-learning is a simple yet quite powerful algorithm and in a sense created a cheat-sheet for the agent to determine the actions that need to be performed. However, if the environment has too many states, in the order of tens of thousands, and also actions in

their thousands, then the cheat-sheet created would simply be too long to process. This presents 2 problems that need to be resolved:

1. The cost required to save and update that table would increase as the number of states increases.

2. The time required to explore each state to create the required Q-table would be unrealistic.

To tackle this, DeepMind technologies came up with an idea to compute, process, and allocate Q-values utilizing neural networks that were already being used to tackle other large prediction problems. This is precisely what Deep Q-Learning or Deep Q-Networks does. [8]

The below diagram accurately illustrates what exactly the DQN algorithm does by replacing the Q-Learning table:

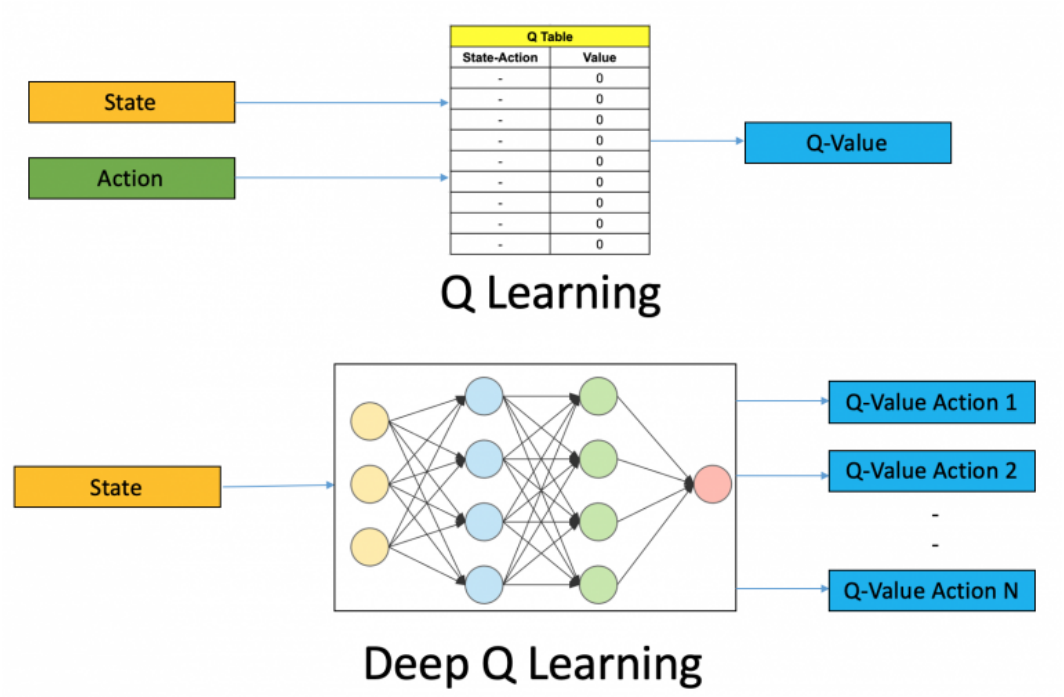

Figure 1.3: Deep Q-Learning generates the Q-value as well as the Actions that need to be taken

As observed in the figure, the two main extensions in DQN are as follows:

1. The action is calculated based on the maximum output of the Q-network

2. The Q-value updated is calculated based on the derivation of the Bellman equation and is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

Here, the section in green represents the target. As seen, this is similar to the Q-learning, Q-value calculation. However, here the prediction is arbitrary depending on the 'R' value and converges to the final Q-value based on newly acquired information. [5]

2 Motivation

Reinforcement learning (RL) continues to be less valuable for business applications than supervised learning, and even unsupervised learning. It is successfully applied only in areas where huge amounts of simulated data can be generated, like robotics and games. However, many experts recognize RL as a promising path towards Artificial General Intelligence (AGI), or true intelligence. Thus, research teams from top institutions and tech leaders are seeking ways to make RL algorithms more sample-efficient and stable [9].

RL has been successfully applied to many different fields such as helicopter control [10], traffic signal control [11], electricity generator scheduling [12], water resource management [13], playing relatively simple Atari games [14] and mastering a much more complex game of Go [15], simulated continuous control problems [16], [17], and controlling robots in real environments [18]. The most notable accomplishment is specifically the use of DQN in playing Atari games by DeepMind technologies which in some cases achieve superhuman capabilities [8].

There has also been an overwhelming development of various Datasets, technologies, Simulation environments in the field of Autonomous Navigation and Driving as seen in the table below:

Dataset	**Description**
Berkeley Driving Dataset (Xu et al., 2017)	Learn driving policy from demonstrations
Baidu's ApolloScape	Multiple sensors & Driver Behaviour Profiles
Honda Driving Dataset (Ramanishka et al., 2018)	4 level annotation (stimulus, action, cause, and attention objects) for driver behavior profile.
Simulator	**Description**
CARLA (Dosovitskiy et al., 2017)	Urban Driving Simulator with Camera, LiDAR, Depth & Semantic segmentation
Racing Simulator TORCS (Wymann et al., 2000)	Testing control policies for vehicles
AIRSIM (Shah et al., 2018)	Resembling CARLA with support for Drones
GAZEBO (Koenig and Howard, 2004)	Multi-robo simulator for planning & control

Figure 2.1: A collection of simulations and datasets to evaluate Autonomous Navigation Algorithms [19]

In addition to these developments and recent work in the field, another project was carried out last year in the Module Distributed Industrial Control Systems at the Institute for Industry Grade Networks and Clouds in the Faculty for Electrical Engineering and Computer Science (IV) at the Technical University of Berlin during the last year. In the project, they have used Unity, ROS, Gazebo, and 2 CNN approaches along with DQN, arriving at promising results. Therefore, this proved to be our main motivation as an opportunity to build on these technologies and explore the possibilities of better results even further.

3 Problem Definition and Scope

The main aim of this project is to develop a Deep Reinforcement Learning code setup for Autonomous Robot Navigation with the latest Open Source Tech-stack available, conduct training, and document the results. The main area of focus is the autonomy of the system, where the environment is unknown and there is no prior data that the agent can rely on during the early stages of training.

Here, the robot is expected to rely solely on sensor data based on the capabilities of the Robot. It is also expected that the training is to be carried out on a simulation environment, and the trained model could be utilized on a real-world robot that exhibits the same capabilities as the robot in the simulation.

Furthermore, the scope of technologies to be utilized are extensions of ROS and Gazebo in the form of their latest versions, and the algorithms that need to be applied fall under the Deep Reinforcement Learning domain.

Finally, the code developed for the training process should be easily scalable, modifiable, adapts to the training requirements, and alters the extent of the algorithm's functionality. The code setup should ideally utilize the latest Open Source Tech-stack available for Machine Learning, specifically Deep Reinforcement Learning.

4 Tech-stack

The structure of the project consists of the use and combination of different frameworks, which as stated in the previous section, are to be extensions of ROS and Gazebo. The simulation, on the other hand, has to be based on a real robot that exists, due to which, Turtlebot3 was chosen. The organization that developed the robot, Robotis, provides examples of use cases and recommends various implementations, library pack-

ages, and simulation technologies around the Turtlebot3 framework. Therefore ROS2, Gazebo9, Turtlebot3, and Keras with a Tensorflow backend have been chosen to form the Tech-stack utilized for this project. In the following section, these frameworks will be introduced briefly and an extension of main functions in comparison to the previous versions will be elucidated.

4.1 ROS2

The Robot Operating System (ROS) is a widely used framework for robotics applications. It has a wide distribution that comes along with comprehensive documentation and many freely available use cases that make it easy to learn. Its successor ROS2, more precisely its version Dashing Diademata, which was released in November 2019, contained some new features that made the use of this newer version more significant compared to ROS.

Robot Operating System (ROS) has long been one of the most widely used robotics middleware in academia and sparingly in the industry. While the huge robotics community has been contributing to new features for ROS 1, since it was introduced in 2007, the limitations in the architecture and performance led to the conception of ROS 2 which addresses these issues.

The software stack for any robot platform needs several software tools like hardware drivers, networking modules, communication architecture, and several robot algorithms. ROS has all these tools under one umbrella, which makes the development of code in the robot platform relatively straightforward. ROS is, however, more than just a middleware, and the availability of various solutions and packages for robot navigation, perception, control, motion planning, simulation, and more makes ROS an important and crucial asset to the field of Robotics.

However, ROS is far from perfect. There are many requirements in today's world that ROS fails to accommodate. A few of the areas that ROS fails are listed below:

- ROS does not support multiple robots with the same master node.
- ROS inherently does not support real-time operation and thus not preferred for time-critical applications.
- ROS needs high-compute resources and network connectivity on-board for the best performance.
- Package management on deployed robots is limited.

- Monitoring, logging, analytics, and maintenance tasks for multiple robots are difficult in commercial settings.
- Multi-robot/fleet management and interaction is not possible. [20]

The main reasons why ROS would impact this project would have to be due to the high computation requirements in training, efficient deployment needs to incorporate the project into a real-world Turtlebot3 robot, and the general potential applications of Autonomous Navigation in real-time systems.

For these reasons, ROS 2 was introduced with a revamped architecture and enhanced features and is being rapidly adopted in the robotics community. It is still in its infancy and several companies and developers have been contributing towards porting the existing packages to ROS 2 compatibility.

All the differences mentioned, justify the use of ROS2 over ROS for almost any application in the field. For this project, we specifically noted the use of Python 3 which includes the latest implementation and upgrades for TensorFlow and Pytorch, the use of other build systems, the efficient use of multiple varied packages and independent installation, the OS compatibility and the ability to pass arguments easily into the roslaunch files.

4.2 Gazebo9

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Additionally, all the Gazebo versions are also Open Source.

Gazebo version 9 was released in 2018. Gazebo9 will have long term support until 2023. The main extensions of this version are the correction of several errors and issues that occurred in the previous version, as well as the addition of the Ignition project into Gazebo thereby exposing several libraries for Math, Transport, and Physics improving the overall performance of the system. It was also transformed from a monolithic project to a broader approach involving several organizations and communities that actively develop and support the platform. It also has added support for easy and quick integration with ROS2. [21]

The Gazebo9 version also has an extension of the SDFormat (Simulation Description Format) and the '.world' files. It is an XML format that describes objects and environ-

Table 4.1: Differences between ROS and ROS2 [20]

ROS	**ROS2**
Uses TCPROS (custom version of TCP/IP) communication protocol	Uses DDS (Data Distribution System) for communication
Uses ROS Master for centralized discovery and registration. Complete communication pipeline is prone to failure if the master fails	Uses DDS distributed discovery mechanism. ROS 2 provides a custom API to get all the information about nodes and topics
ROS is only functional on Ubuntu OS	ROS 2 is compatible with Ubuntu, Windows 10 and OS X
Uses C++ 03 and Python2	Uses C++ 11 (potentially upgradeable) and Python3
ROS only uses CMake build system	ROS 2 provides options to use other build systems
Has a combined build for multiple packages invoked using a single CMakeLists.txt	Supports isolated independent builds for packages to better handle inter-package dependencies
Data Types in message files do not support default values	Data types in message files can now have default values upon initialization
roslaunch files are written in XML with limited capabilities	roslaunch files are written in Python to support more configurable and conditioned execution
Cannot support real-time behavior deterministically even with real-time OS	Supports real-time response with apt RTOS like RTPREEMPT

ments for robot simulators, visualization, and control. Originally developed as part of the Gazebo robot simulator, SDFormat was designed with scientific robot applications in mind. Over the years, SDFormat has become a stable, robust, and extensible format capable of describing all aspects of robots, static and dynamic objects, lighting, terrain, and even physics. The components for this project, including obstacles and other objects were in the form of SDF files.

All aspects of a robot can be accurately described using SDFormat, whether the robot is a simple chassis with wheels or a humanoid. In addition to kinematic and dynamic attributes, sensors, surface properties, textures, joint friction, and many more properties can be defined for a robot. These features allow you to use SDFormat for simulation, visualization, motion planning, and robot control. The simulation requires rich and com-

plex environments in which models exist and interact. SDFormat provides the means to define a wide variety of environments. In Gazebo9, these are achieved in terms of '.world' file extensions. [22]

Gazebo9 seems to be the most stable and efficient version currently with the integration of ROS2 and community support. Furthermore, Gazebo allows Turtlebot3 to use virtual sensor data in the simulator: IMU, LDS, and camera information, and the simulation of Turtlebot3 with SLAM or Navigation2. Since we intended to depend on solely sensor data for this project, Gazebo9 made this specific task of simulation more efficient.

4.3 Turtlebot3

There are 3 versions of the TurtleBot series. In 2017, TurtleBot3 was developed with features to supplement the lacking functions of its predecessors, and the demands of users. TurtleBot3 is a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping.

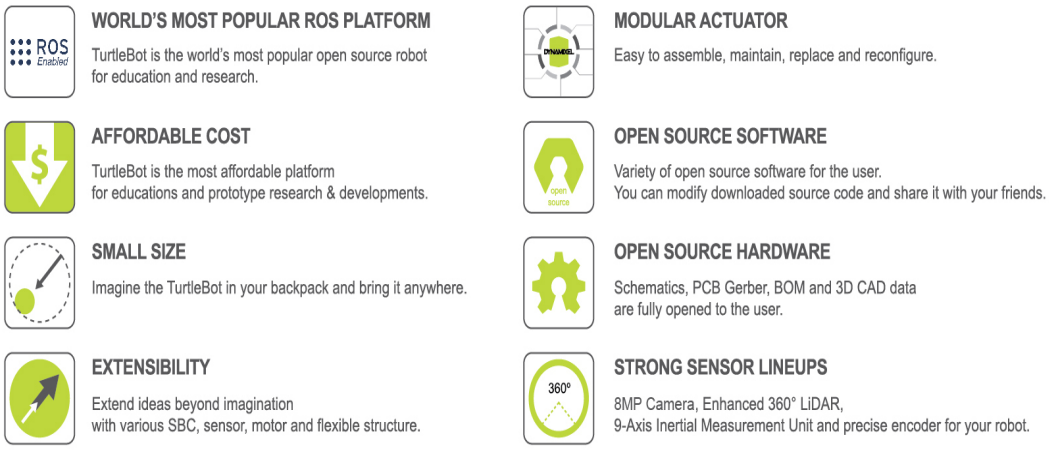

Figure 4.1: Salient features of Turtlebot3 [23]

The goal of TurtleBot3 is to dramatically reduce the size of the platform and lower the price without having to sacrifice its functionality and quality, while at the same time offering expandability. The TurtleBot3 can be customized into various configurations depending on how you reconstruct the mechanical parts and use optional parts such as the computer and sensor. Also, TurtleBot3 evolved with a cost-effective and small-sized SBC that is suitable for a robust embedded system, 360-degree distance sensor, and 3D printing technology. The TurtleBot3's core technology is SLAM, Navigation, and Manipulation, making it suitable for home service robots. The TurtleBot can run SLAM(simultaneous localization and mapping) algorithms to build a map and can drive

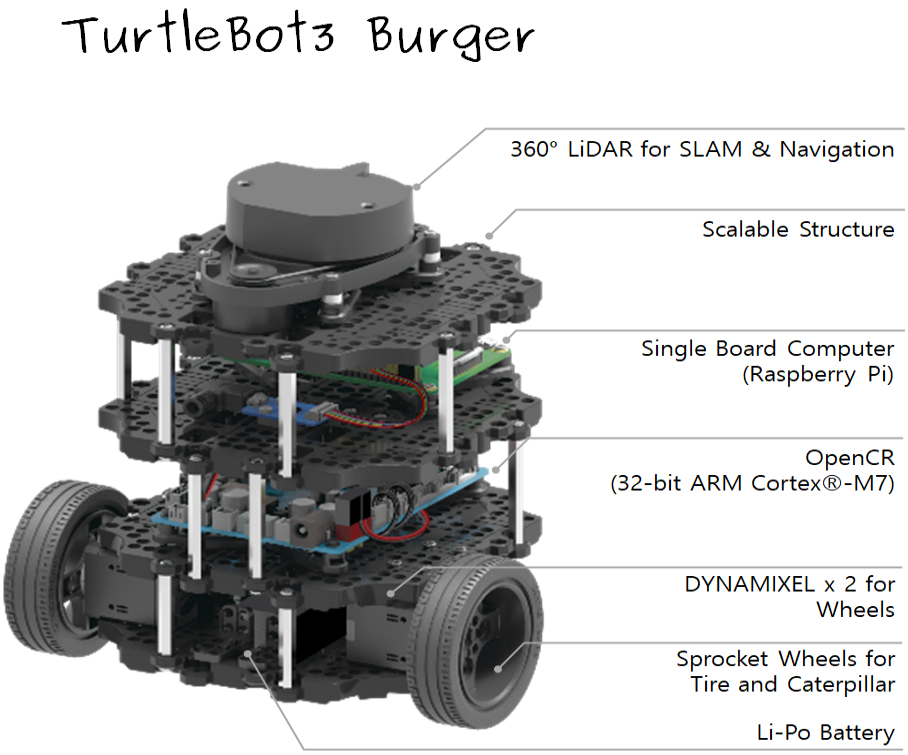

Figure 4.2: Turtlebot3 Burger components [23]

around your room.

In this project, we utilize mainly the LiDAR component of the Turtlebot3 Burger, for Laser Distance Sensing, to sense the obstacles that appear before the robot. The only other components that we may need to consider are the DYNAMIXEL wheels for motion, and Raspberry Pi and Open CR components for processing the information that is relayed to the robot.

Robotis offers machine learning packages for efficient integration of Turtlebot3, gazebo9 simulation platform, and the Machine Learning logic for Autonomous Navigation. These packages in turn use Tensorflow and Keras to expose various options to use and configure Deep Reinforcement Learning algorithms within the simulation environment and store them as models. These can then be used for the actual robot in the real world.

4.4 Tensorflow and Keras

TensorFlow is an end-to-end open-source platform for machine learning. It's a comprehensive and flexible ecosystem of tools, libraries, and other resources that provide workflows with high-level APIs. The framework offers various levels of concepts to build and deploy machine learning models. Some of the features of Tensorflow are described

below:

- Easy Model Building: TensorFlow offers multiple levels of abstraction to build and train models.
- Robust ML Production Anywhere: TensorFlow lets you train and deploy your model easily, no matter what language or platform you use.
- Powerful Experimentation For Research: TensorFlow gives you the flexibility and control with features like the Keras Functional API and Model Subclassing API for creation of complex topologies.[3] [24]

Keras, on the other hand, is a high-level neural networks library that is running on the top of TensorFlow, CNTK, and Theano. Using Keras in deep learning allows for easy and fast prototyping as well as running seamlessly on CPU and GPU. This framework is written in Python code which is easy to debug and allows ease for extensibility. The main advantages of Keras are described below:

- User-Friendly: Keras has a simple, consistent interface optimized for common use cases which provides clear and actionable feedback for user errors.
- Modular and Composable: Keras models are made by connecting configurable building blocks, with few restrictions.
- Easy To Extend: With the help of Keras, you can easily write custom building blocks for new ideas and researches.
- Easy To Use: Keras offers consistent simple APIs which helps in minimizing the number of user actions required for common use cases, also it provides clear and actionable feedback upon user error. [25] [24]

In this project, we use the turtlebot3_dqn packages which use the DQN models within Keras and TF-Agents which are used for training a model, generation of Q-values, and corresponding action that the Agent needs to take based on the output of the network. It also exposes various files and classes that help us set various parameters and hyperparameters that will be used in the training process, and will be discussed in the following sections.

5 Implementation

To begin implementing the project, we were first required to understand the system architecture and conceptualize the interactions between the various technologies involved.

The following figure presents the dataflow between various systems and the functionalities of each component. Our main task would then be to actualize these functions through the code setup, installation, and the use of various ROS2, Gazebo9, and Turtlebot3 packages.

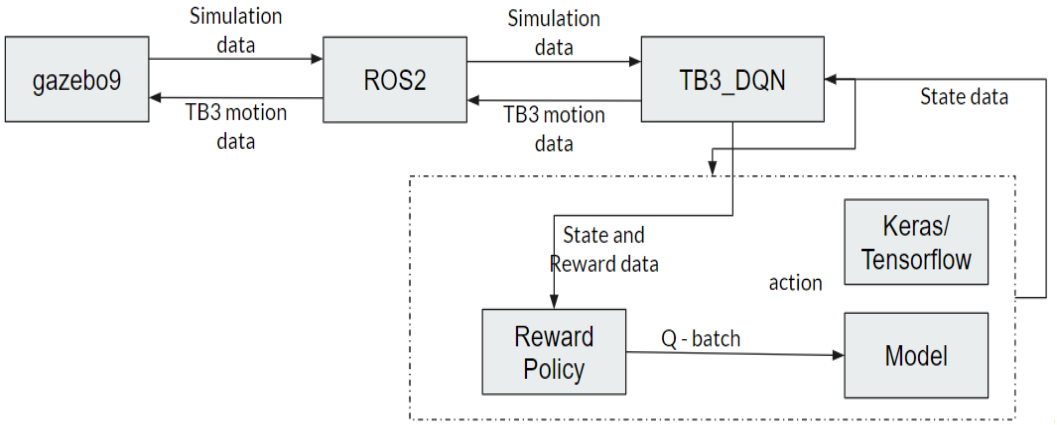

Figure 5.1: System Diagram showing the dataflow and Interaction of various components involved in this project

In this system, the simulation portrays a real-world environment. The turtlebot3 burger robot exists within a gazebo9 simulation. The simulation data is constantly being communicated between ROS2 (which acts as an Operating System) and the turtlebot3_dqn packages, which is the hub of all data and DQN processing. The simulation data consists of mainly the State data for each time step, within the current episode. This State data is utilized by various components that use the Tensorflow and Keras libraries for various functions. The functions are performed by the DQN model and the Reward Policy calculations are executed based on the Q-batch memory. The main output from this system is the Action that the Turtlebot3 burger robot needs to take, which is then accordingly relayed back to the ROS2 and gazebo9 platforms. The various specifics of the Action, Reward, Batch, and State data will be discussed in the following sections.

5.1 Installation process

The various components that are required, have now been laid out in the System diagram. Each of these technologies has to then be installed to start setting up the simulation and the DQN model for training. The first step would be to set up the ROS2 platform. For this, as mentioned in the previous chapter, we have chosen the ROS2 Dashing Diademata version and have followed the installation steps for the Debian packages as mentioned in [26]

The next step is to install the gazebo9 simulation platform. For this, we first need to follow the installation guidelines specified in [27]. However, we need to follow the step-by-step guide as we need to interrupt the installation of gazebo11, as turtlebot3 still does not support stable versions of DQN in gazebo11 and instead install gazebo9 during the final stages of installation.

The next stage would be to follow the various steps of installation of turtlebot3 specific packages mentioned in [23] as follows:

- The ROS2 dependency packages were first installed under the ROS2 setup page
- In the same page, the turtlebot3 packages were installed and the Bash commands for Setup were saved
- The turtlebot3_burger model was brought up by following the instructions on the ROS2 Bringup page. This was mainly to test whether the roslaunch functionality was able to integrate ROS2, gazebo9, and turtlebot3 packages effectively.
- The turtlebot3_gazebo packages were installed following the steps mentioned in the ROS2 Simulation page. We then launched various worlds that are provided in these packages to check whether the packages function.
- The python 3.6 dependency packages were then installed following the instructions on the ROS2 Machine Learning page
- The Tensorflow and Keras packages were then installed by following the instructions mentioned on the same page
- The turtlebot3, turtlebot3_msgs, and turtlebot3_simulations packages were then installed following the links provided in the page
- The final step was to install the turtlebot3_machine_learning packages following the steps provided on the same page

5.2 Installation Obstacles

It must be noted that the Installation process did not go as smoothly as expected. Many problems were encountered along the way. The following issues were noted:

- Although, one of the main advantages of using ROS2 is that it supports its use in multiple OSs, we came across several issues with the setup on Windows and had to revert to using Linux OS. Similar problems with OS compatibility were encountered with the gazebo9 packages and the subsequent packages for turtlebot3 integration of simulation and Machine Learning packages.

- We also noticed that the steps mentioned in the 'Installation process' section above, need to be meticulously followed in the same order. They were not mentioned in any order in the documentation provided, therefore led to numerous issues with the debugging process.
- A few packages that are installed for turtlebot3 does not support previously installed versions of Python 3.6 supported packages. Therefore a few packages needed to be carefully reverted to the previous versions or excluded from the installation process altogether.
- The turtlebot3 projects for Simulation, Tensorflow, Keras, and Machine Learning packages directed us to the latest version of the code on the Git repositories. However, we discovered issues with the package code after December 2019 and had to revert each cloned code to the previous stable version accordingly.
- Finally to successfully install all packages and run the code, computers of relatively high processing speeds were required. The installation process failed on all but one laptop in our possession that had 16GB RAM, an Intel i7 8th gen processor, and used SSDs for storage. If the installation did not carry forward, the same process of installation had to be repeated in the same order from start to finish.

On overcoming the issues mentioned above, the code setup from then on for the DQN algorithm was relatively easy. And the simulation worked without any during the training process and while recording the results.

5.3 DQN Algorithm

Now that the installation and setup have been completed, the next step would be to code the logic for the DQN algorithm. Here, several components are conceptualized to split the various functionalities as given in the figure below:

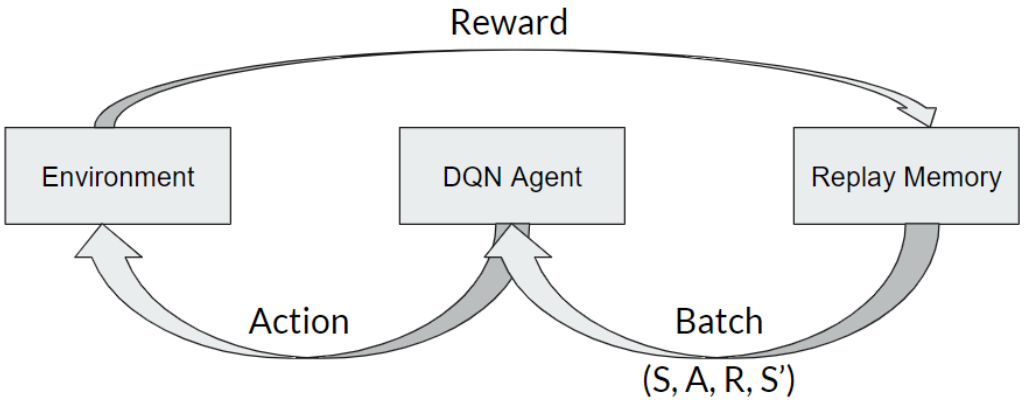

Figure 5.2: Interaction of DQN Agent with other components within the DQN Logic

As seen in the figure, the DQN Agent is central to the Logic. It represents the turtlebot3_burger robot in the simulation. Here the DQN agent is expected to perform an Action in the Environment, therefore the Action data needs to be set to relay the data to the simulation. The action will then lead to a consequential State in the Environment. Based on the current State, the reward needs to be calculated by the DQN Algorithm and stored in the Replay Memory. Here next course of action is determined, and is padded with the current state, the next state as well as the Reward and stored as a Batch. Based on the sequence and various hyperparameters the next Batch will be selected that will relay the next Action that needs to be taken by the DQN Agent in the Environment. This cycle repeats for each time step within each episode. DQN algorithm will thereby attempt to keep increasing the Cumulative Reward of the entire system.

5.3.1 Set State

A state is an observation of an environment and describes the current situation. In this project, we have used LDS to obtain information about the current state. Here, the size of the state is set to 26, out of which 24 values pertain to the LDS data, one for the distance to the goal, and another for the angle to the goal. By default, the Turtlebot3's LDS value is set to 360. This can be changed in a xacro file in the turtlebot3 description directory as shown below:

```
<xacro:arg name="laser_visual" default="false"/>   # Visualization of LDS. If you want to see LDS, set to `true`
```

```
<scan>
  <horizontal>
    <samples>360</samples>            # The number of sample. Modify it to 24
    <resolution>1</resolution>
    <min_angle>0.0</min_angle>
    <max_angle>6.28319</max_angle>
  </horizontal>
</scan>
```

Figure 5.3: Information in the turtlebot3 burger description xacro file

In this project however, we have chosen the default LDS value as it could more accurately relay information about obstacles around and in close proximity.

5.3.2 Set Goal

The goal is configured as a simple translucent object in the SDF file for each world. We utilized a red shade, and a circle or square that appears within the bounds of the walls, and does not coincide with the walls. For this project, we have used a simple random generator as shown in the code below. A list of arbitrary goal positions is already set based on the conditions stated concerning the obstacles. The indices are arbitrarily

generated for each episode based on which the x and y coordinates are chosen. The choice of which position to select is randomly generated as a result.

```
goal_pose_list = [[1.0, 0.0], [2.0, -1.5], [0.0, -2.0], [2.0, 2.0], [0.8, 2.0],
                  [-1.9, 1.9], [-1.9, 0.2], [-1.9, -0.5], [-2.0, -2.0], [-0.5, -1.0]]
index = random.randrange(0, 10)
self.goal_pose_x = goal_pose_list[index][0]
self.goal_pose_y = goal_pose_list[index][1]
```

Figure 5.4: Random goal position generator

5.3.3 Set Action

The DQN-Agent is now supposed to reach these randomly chosen goals. To do so, it can choose between a set of five actions. Since it is initialized with a constant linear velocity of 0.15 m/s, it can only change the angular velocity, which is in the beginning 0 (Action 2). Then it can choose between continuing with the present angular velocity or changing it by a given value as can be seen in the following table:

Action	Angular velocity (rad/s)
0	-1.5
1	-0.75
2	0
3	0.75
4	1.5

Table 5.1: Actions of the DQN-Agent

5.3.4 Reward Policy

The Reward policy is determined by 2 main factors. Firstly, the reward depends on the current state of the turtlebot3 and the goal, i.e. the distance between the turtlebot3 and the goal which is given by the following formula:

$$r_{distance} = \frac{2 \cdot goaldistance_{init}}{goaldistance_{init} + goaldistance} - 1,$$

and the angle between the current normal axis of turtlebot3's Action and the goal position, which is given by:

$$r_{yaw} = 1 - 2 \cdot \sqrt{\frac{goalangle}{\pi}}$$

The reward also depends on the distance between the turtlebot3 and the obstacle at any given point, so as to discourage a collision, and is as given below:

$$r_{obstacle} = \begin{cases} -2, & if\ obstacledistance_{min} < 0.25 \\ 0, & else \end{cases}$$

Therefore, the total reward is calculated as follows:

$$r_{total} = r_{yaw} + r_{distance} + r_{obstacle}$$

A slight boost is additionally given depending on whether the episode ended in success or failure, where an episode is successful if the robot reaches the goal and an episode fails when the robot collides with an obstacle or the episode has timed out, and is as given below:

$$r_{total} = \begin{cases} if\ success & r_{total} + 5 \\ else & r_{total} - 10 \end{cases}$$

6 Training

The training process is what verifies the feasibility of this project. However, before we start the training process, the learning process of the DQN algorithm needs to be optimized. This has been accomplished by setting the Hyperparameters for the DQN files. The name of each parameter and their significance is showcased below:

Hyper parameter	default	description
episode_step	6000	The time step of one episode.
target_update	2000	Update rate of target network.
discount_factor	0.99	Represents how much future events lose their value according to how far away.
learning_rate	0.00025	Learning speed. If the value is too large, learning does not work well, and if it is too small, learning time is long.
epsilon	1.0	The probability of choosing a random action.
epsilon_decay	0.99	Reduction rate of epsilon. When one episode ends, the epsilon reduce.
epsilon_min	0.05	The minimum of epsilon.
batch_size	64	Size of a group of training samples.
train_start	64	Start training if the replay memory size is greater than 64.
memory	1000000	The size of replay memory.

Figure 6.1: Hyper Parameters that need to be set for Learning optimization

These are the default values that are set in the turtlebot3_dqn packages and these are the same values that we have used in this project to obtain the needed results. The presence of the epsilon value here suggests that the model is in fact epsilon greedy and will set arbitrary Actions to avoid loopholes in the robot's states and the training process. The epsilon value ranges from 1 at the beginning of the training process and periodically decreases over time based on the epsilon_decay value. The least possible epsilon value is seen as 0.05.

To begin the training we also need to set up the environment. Here we have chosen the following two worlds:

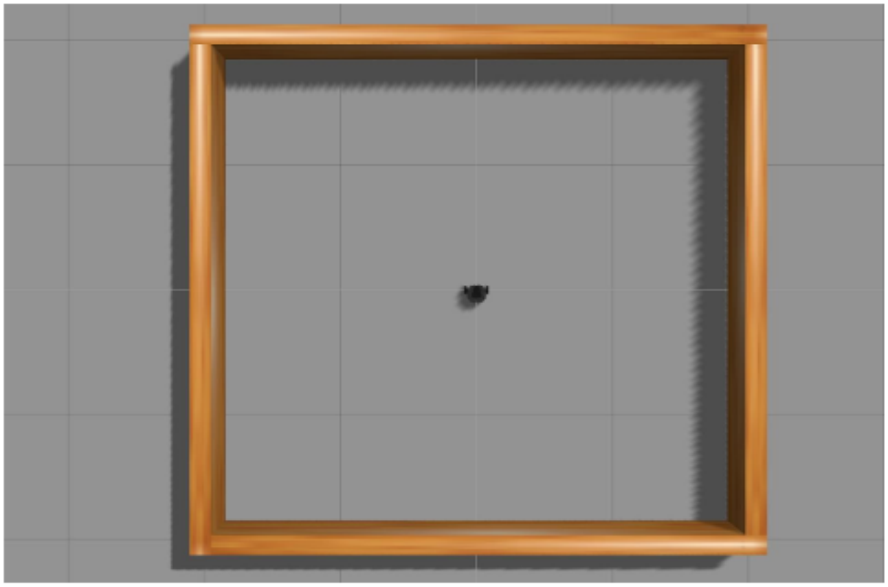

Figure 6.2: World with 4 walls

It is to be noted that the walls are however considered as obstacles, the goals are generated only within the walls and not generated on top of them.

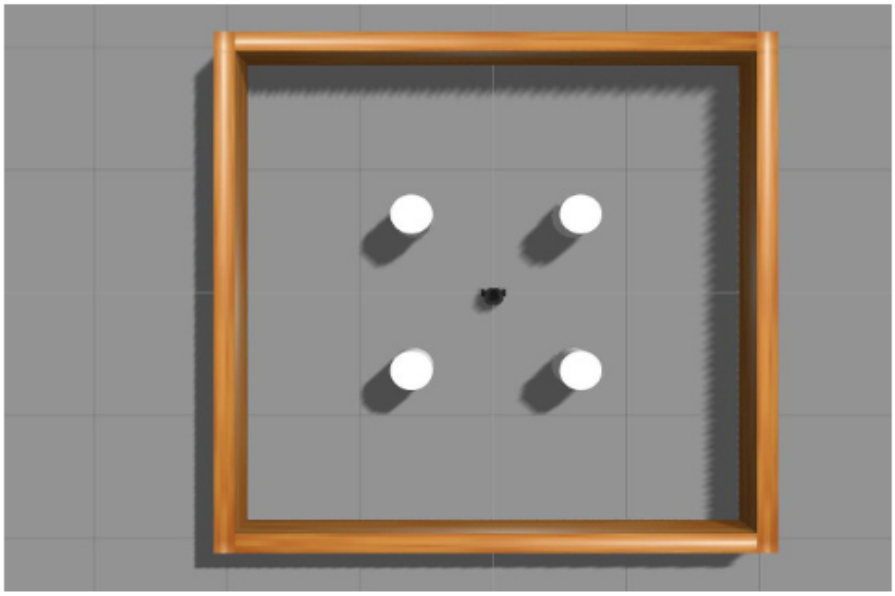

Figure 6.3: World with 4 walls and 4 obstacles

The walls are considered as obstacles in addition to the 4 cylindrical obstacles enclosed by the walls. The goals are generated only within the walls and should not coincide with the walls. However, they may be generated on top of the cylindrical obstacles.

Now to start the training, we run the ROS2 launch file, then the file with the gazebo dependencies, followed by the file with the environment interactions linked to the SDF files, and finally the Agent file. Each of these files needs to be run on separate terminals. Several code snippets were written within each file to record the output and generate the log files simultaneously, to obtain the results. The training was also carried out using one laptop with the capabilities mentioned in chapter 5.

7 Results

The results were observed in two phases. The first phase involved manual observations during the training process to make sure that it was progressing as intended. The second phase involved studying the log files more closely and observing patterns after the training process has been completed. This was carried out to discover options that could positively alter results, which would prompt variation in training methods.

As a result, it was observed that with both the worlds, the training began frantically, with arbitrary choices of actions. And along the mid-way, the actions slowly began to stabilize. This involved at least a minimal avoidance of obstacles, even if the goal was not reached. This progressed to episodes where the avoidance of obstacles was at a higher rate and the goal was being reached for multiple episodes consequentially. At this stage, the training was stopped and the results were noted. A collection of observations have been plotted into graphs and have been mentioned in the following sections.

7.1 Results - World 1

The first criterion observed was the training time needed for the completion of each episode, i.e., for them to end in either success, failure, or a timeout. As seen initially, the episodes are short, usually due to frequent failures in the early stages.

From around episode 70 to around 130, the epsilon values are relatively high and the Replay memory has already enabled the agent to avoid obstacles. Therefore at this stage, the robot explores the most, resulting in increased episode times.

However, after around 140 episodes, the model has learned that the most effective way to increase cumulative reward is to navigate directly to the target. It could also be due to the increase in the number of successes at this stage. Further results, will exhibit this pattern more clearly.

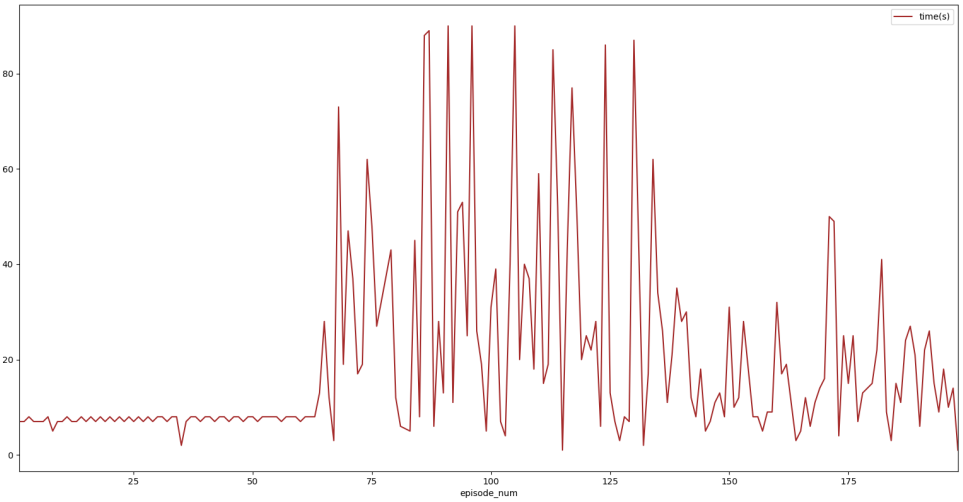

Figure 7.1: Training time per episode - World 1

The next 2 plots are more veritable than observations. The epsilon values as defined in the hyperparameters, decrease from 1 to around 0.05. And the Replay memory increases alongside the total number of episodes, as the model stores increasing amounts of Batch information for processing.

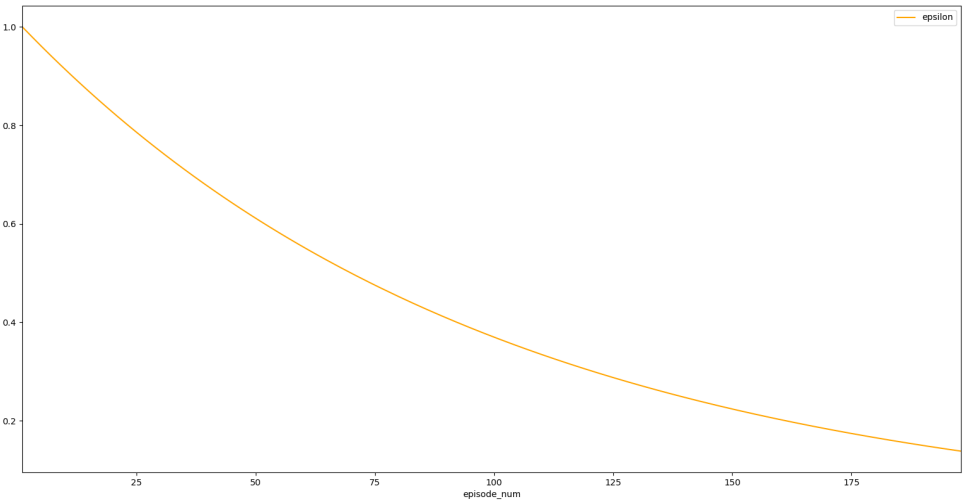

Figure 7.2: Epsilon value in each episode - World 1

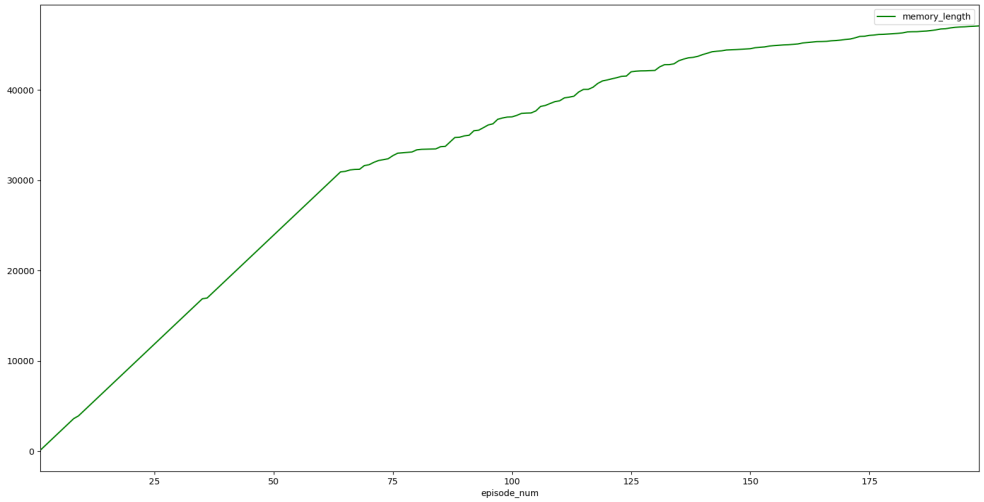

Figure 7.3: Memory used for each episode - World 1

The next plot shows how the Cumulative reward is increasingly negative at the start, and converges to 0, indicating that the cumulative reward has switched to increasingly positive and has effectively neutralized the negative results from the initial stages. Meanwhile, the epsilon value decreases, from 1.0 to 0.05 as expected. This shows that with a higher epsilon value, the agent is expected to make a higher rate of arbitrary moves, which in turn, adversely affects the cumulative reward.

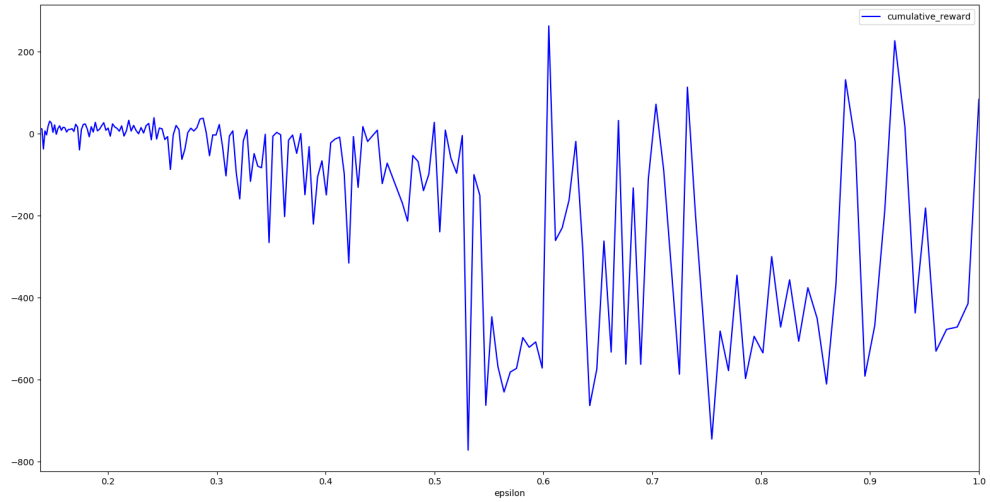

Figure 7.4: Epsilon value and the corresponding cumulative reward - World 1

The next plot shows the cumulative reward stabilizing overtime against the episode number. It also should be noted that at around 150 episodes the Model already shows positive results, which have been confirmed by manual observations of the agent's performance at this stage.

At this point, the agent is already successfully navigating towards the goal without any collisions or timeouts. Therefore successful results were observed repeatedly despite a relatively low number of training episodes.

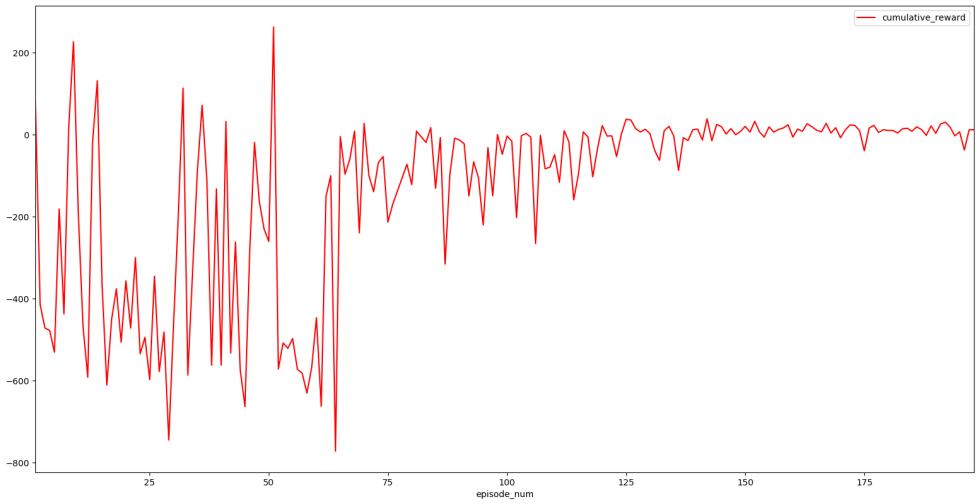

Figure 7.5: Epsilon value and the corresponding episode number - World 1

7.2 Results - World 2

The same approach was carried out for noting observations for the second world as well. Again, as seen initially the episodes are short, usually due to immediate failures in the early stages. From around episode 70 to around 230, the epsilon values are relatively high and the Replay memory has already enabled avoiding obstacles.

Therefore at this stage, the robot explores the most, resulting in increased episode times. However, after around 250 episodes, the model has learned that the most effective way to increase cumulative reward is to navigate directly to the target. It could also be due to the increase in the number of successes at this time.

Further results, will evidently shed some light to this pattern. Therefore we stopped the training at around 280 to 300 episodes to manually observe and record results.

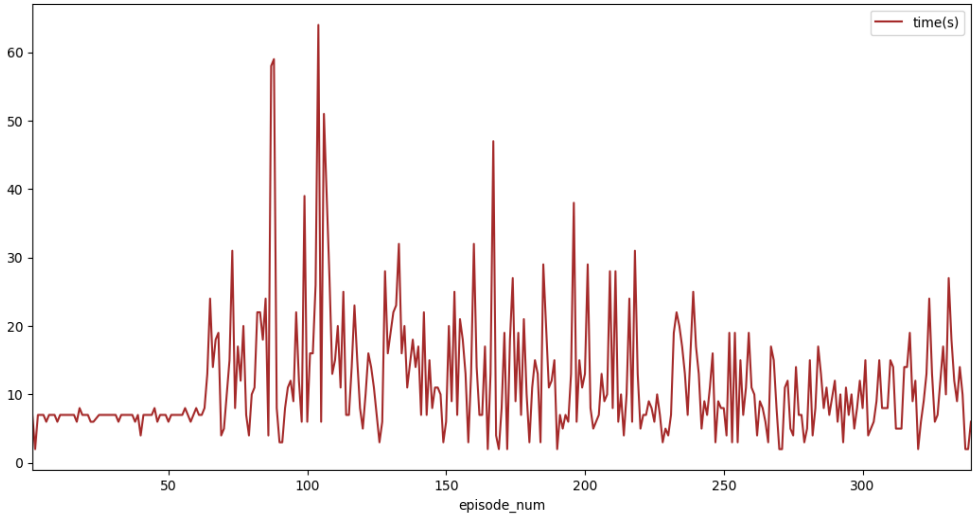

Figure 7.6: Training time per episode - World 2

Similar to the previous world, the next 2 plots are more veritable than observations. The epsilon values as defined in the hyperparameters, decrease from 1 to around 0.05. And the Replay memory increases with the number of episodes as the model learns more and more Batch information to keep track of.

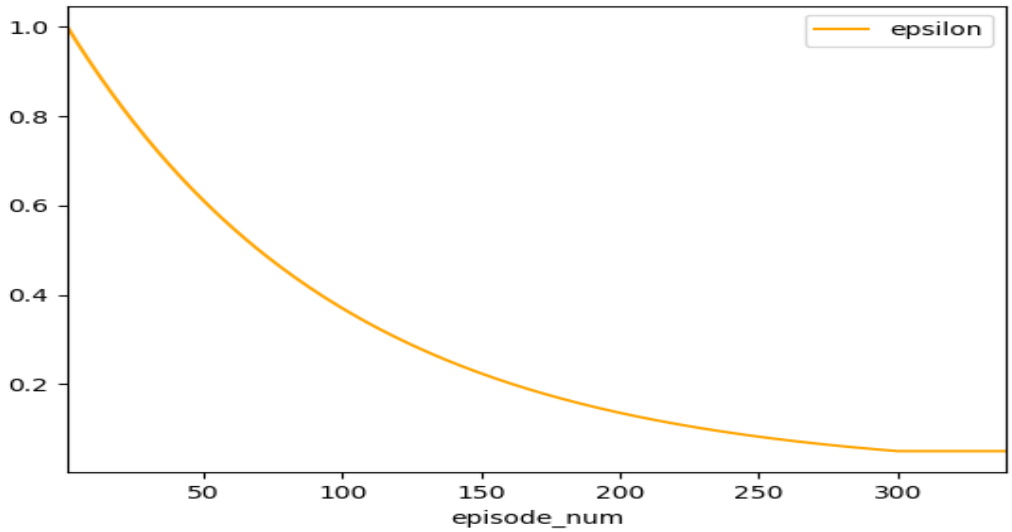

Figure 7.7: Epsilon value in each episode - World 2

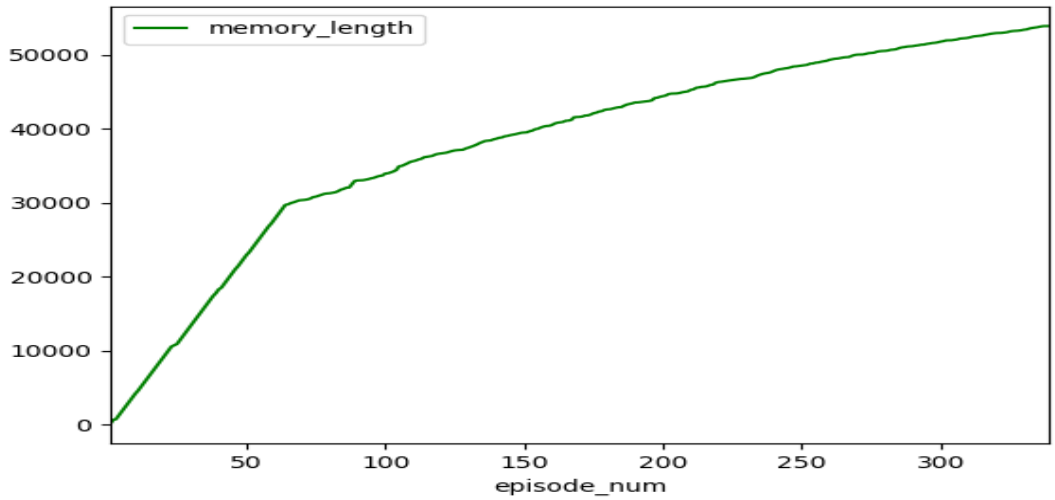

Figure 7.8: Memory used for each episode - World 2

The next plot shows how the Cumulative reward is increasingly negative at the start, and converges to 0, indicating that the cumulative reward has switched to increasingly positive and neutralizes the negative results from the initial stages.

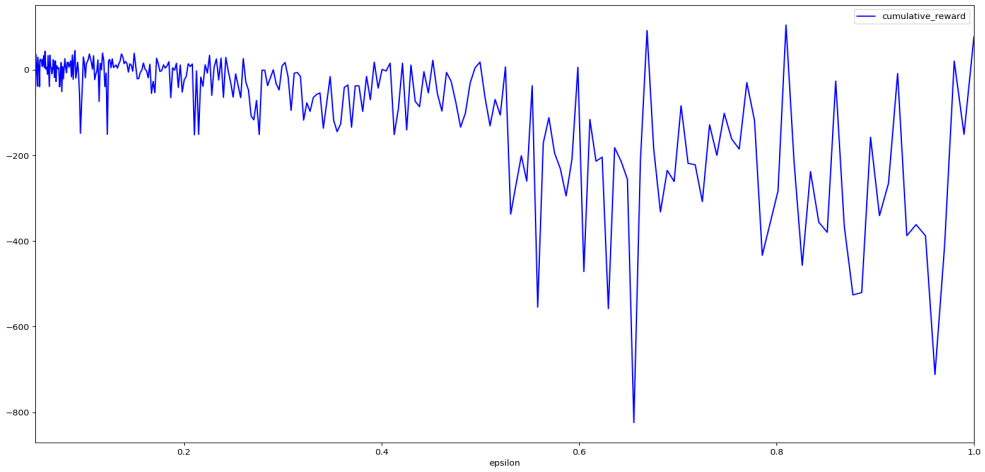

Figure 7.9: Epsilon value and the corresponding cumulative reward - World 2

However, in this world, we can see of slight fallback to old patterns as the epsilon value drops (below 0.2). This also shows that with a high epsilon value, the agent is expected to make a higher rate of arbitrary moves, which also adversely affects the cumulative reward. But here observations suggest that a lower epsilon decay value could help the performance improve gradually, but more efficiently. This also shows that with

a high epsilon value, the agent is expected to make a higher rate of arbitrary moves, which also adversely affects the cumulative reward. But here observations suggest that a lower epsilon decay value could help the performance improve gradually, but more efficiently. The next plot shows the cumulative reward stabilizing overtime against the episode number. It also should be noted that at around 250 episodes the Model already shows positive results, which have been confirmed by manual observations of the agent's performance at this stage.

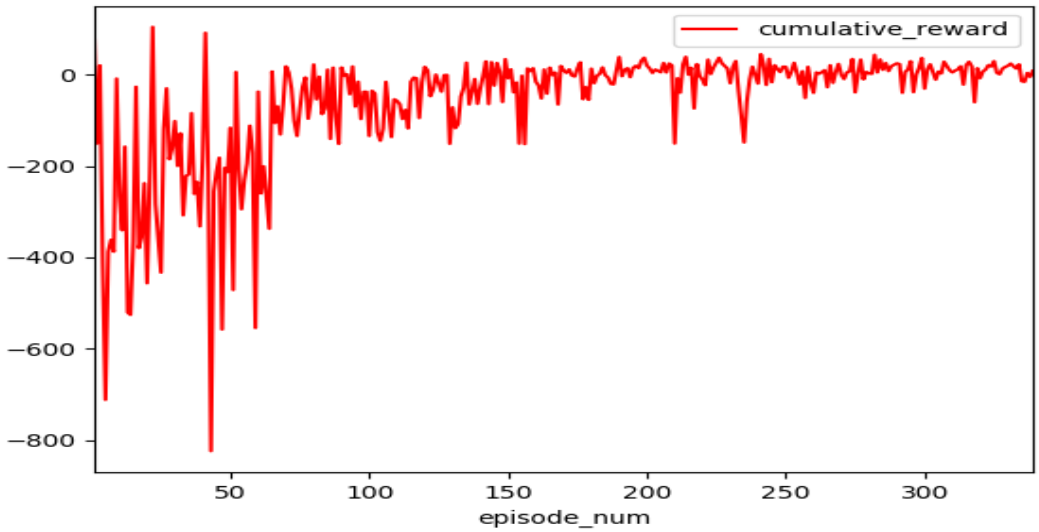

Figure 7.10: Epsilon value and the corresponding episode number - World 2

At this point, the agent is already successfully navigating towards the goal without any collisions or timeouts and is also able to navigate around the obstacles smoothly. Therefore successful results were observed repeatedly despite a relatively low number of training episodes.

In both cases, on making successful manual visual observations, it was noted that the agent performed 10 consecutive steps without failures and therefore the training was stopped and the results were plotted. Due to the epsilon greedy algorithm under DQN, and the powerful turtlebot3 packages for Tensorflow, Keras, and the LDS data utilization in the State memory, the results were obtained at a relatively low number of episodes as previously thought to have been needed.

7.3 Difficulties

One of the main reasons that we were pushed to stop the training, is due to the failure in extending the training beyond a stipulated number of episodes. This was either due to processing limitations of the processor used for the project or lack of stable internet

connectivity. Even if the training were to be stopped, a lot of the core logic needs to be altered to continue the training seamlessly, from a model that was stored. This would require modifying the epsilon and decay values, the number of steps in an episode, perhaps even reduce the learning rate to compensate for lost batches, etc. The following plots illustrate the issues faced more closely:

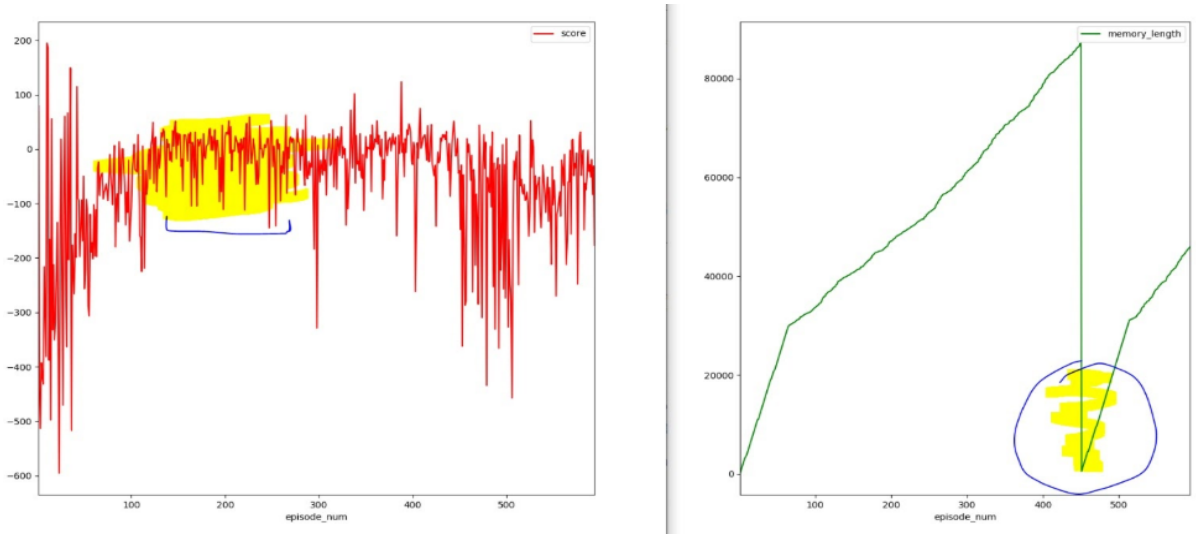

Figure 7.11: Drop in performance as training is interrupted/ Failure to retain relay memory due to lack of library functions to save the batch buffer

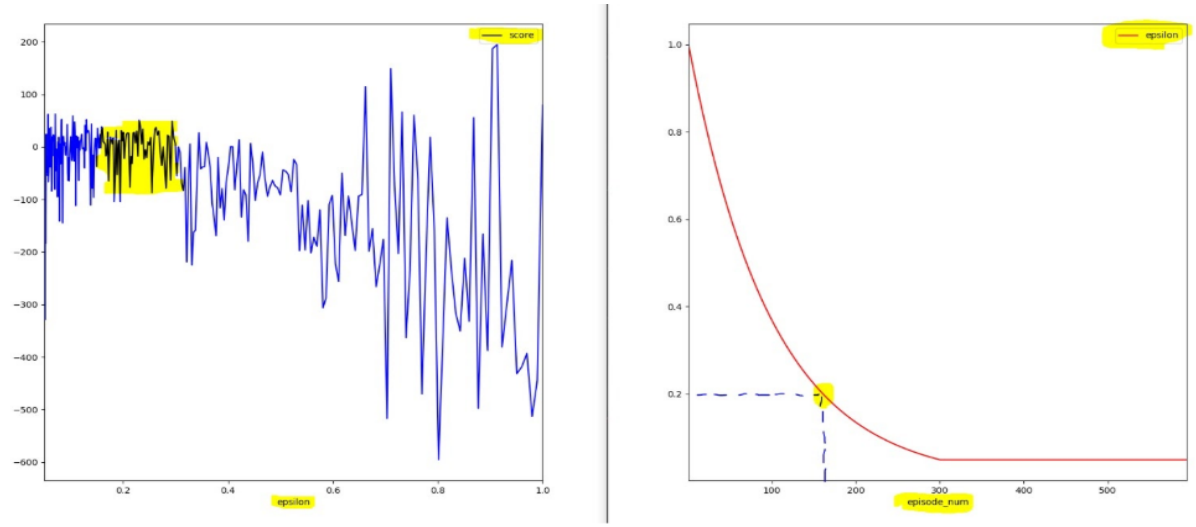

Figure 7.12: Mapping of cumulative reward and epsilon value/ And then mapping the epsilon value to the correct episode number to obtain the window for results

Therefore by observing these limitations, the following method was used to obtain an approximate episode number, where the performance is stable enough. The episode can

then be manually observed and the results can be recorded.

In addition to the two worlds mentioned above, an additional house world was used to attempt the training process and obtain results. However, due to constraints in resources and time, an extension of the 'set goal' logic posed limitations. This would also require a large number of training episodes which would be computationally heavy in comparison to previous attempts made. However, the current model has been applied and tested in the house world without a goal and programmed to avoid obstacles. And this has shown promising results for future implementations of a similar model with extended simulation capabilities.

8 Conclusion

The applications of DQN for Autonomous Navigation is quite varied and vast. However, the latest platforms that are being developed to implement these methods are still in their infancy. During this project, we have come across many obstacles and difficulties. Firstly, during the installation and setup process, numerous issues in the ROS2, gazebo9, and turtlebot3 packages were encountered, including limited version compatibility and the lack of support for newer versions of gazebo including versions 10 and 11. Despite this, once the code was set up, the project was carried out relatively smoothly. This was due to the seamless integration of all turtlebot3 packages with ROS2, gazebo9, Tensorflow, Keras, and the Machine learning libraries. The training showed significant efficiency with good results at a relatively lower number of episodes. The Reward function was recorded, stabilizing after 100 - 200 episodes. As the epsilon value decreases, the agent needs decreasing amounts of time to reach the goal. Therefore, to optimize these results, the project could be attempted with lower epsilon decay values, the minimum epsilon value could be altered to preferably, a little higher, the Learning Rate could be varied, and further results could be recorded. There are numerous ways to tweak just the DQN algorithm alone, which gives us a glimpse of the extent to which other innumerable algorithms can be modified to suit the demands of Autonomous Navigation.

9 Future Work

The future work envisaged for this project could invariably keep shifting, as increasingly many technologies and Deep Reinforcement Learning models are developed, broadening the scope of their applications in Autonomous Navigation. For example, even from a logical standpoint, programmatic changes can be made in the Reward function, or as previously stated, in the epsilon values to obtain varying results. Another opportunity

for work is to increase the current scope, by training the same model, for a much higher number of episodes. This could be carried out by utilizing clusters and computers with better processors for the overall training process within the simulation. Furthermore, the simulation environment could also be extended to involve complex worlds as explored in the previous chapters. However, the main requirement at this stage, for the technologies utilized in this project, is to fix the compatibility issues that exist during the installation and build steps, in packages used, and create an easy-to-launch setup.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 2014, 2015. [Online]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf

[2] *Types of Machine Learning Out There*, IDAP, 2019. [Online]. Available: https://idapgroup.com/

[3] *Introduction to RL and Deep Q Networks*, TensorFlow, 2020. [Online]. Available: https://www.tensorflow.org/agents/tutorials/0_intro_rl

[4] *The Fairly Accessible Guide to the DQN Algorithm*, mc.ai, 2020. [Online]. Available: https://mc.ai/the-fairly-accessible-guide-to-the-dqn-algorithm/

[5] *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*, Analytics Vidhya, 2019. [Online]. Available: https://www.analyticsvidhya.com/

[6] *Reinforcement Learning Demystified: Markov Decision Processes*, Towards Data Science, 2018. [Online]. Available: https://towardsdatascience.com/

[7] F. S. Melo, *Convergence of Q-learning: a simple proof*, Institute for Systems and Robotics, Instituto Superior Técnico, Lisboa, Portugal, 1997. [Online]. Available: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, DeepMind Technologies, 2015. [Online]. Available: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[9] M. Yao, *Breakthrough Research in Reinforcement Learning from 2019*, TopBots, 2019. [Online]. Available: https://www.topbots.com/top-ai-reinforcement-learning-research-papers-2019/

[10] D. K. Naik and R. Mammone, *Meta-neural networks that learn by learning. In Neural Networks*, IJCNN., International Joint Conference on, volume 1, pages 437–442. IEEE, 1992.

[11] P. Mannion, J. Duggan, and E. Howley, *An experimental review of reinforcement learning algorithms for adaptive traffic signal control*, Autonomic Road Transport Support Systems, pages 47–66. Springer International Publishing, 2016.

[12] P. Mannion, K. Mason, S. Devlin, J. Duggan, and E. Howley, *Multi-objective dynamic dispatch optimisation using multi-agent reinforcement learning*, In Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pages 1345–1346, 2016.

[13] K. Mason, P. Mannion, J. Duggan, and E. Howley, *Applying multi-agent reinforcement learning to watershed management*, In Proceedings of the Adaptive and Learning Agents workshop (at AAMAS 2016), 2016.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski, *Human-level control through deep reinforcement learning. Nature*, 518(7540):529, 2015.

[15] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, *Mastering the game of go with deep neural networks and tree search. nature*, 529(7587):484–48, 2016.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, , and D. Wierstra, *Continuous control with deep reinforcement learning*, arXiv preprint arXiv:1509.02971, 2015.

[17] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, *Trust region policy optimization*, In International Conference on Machine Learning, pages 1889–1897, 2015.

[18] S. Levine, C. Finn, T. Darrell, and P. Abbeel, *End-to-end training of deep visuomotor policies*, The Journal of Machine Learning Research, 17(1):1334–1373, 2016.

[19] V. Talpaert, I. Sobh, B. R. Kiran, P. Mannion, S. Yogamani, A. El-Sallab, and P. Perez, *Exploring Applications of Deep Reinforcement Learning for Real-world Autonomous Driving Systems*, U2IS, ENSTA ParisTech, Palaiseau, France; AKKA Technologies, Guyancourt, France; Valeo Egypt, Cairo; Galway-Mayo Institute of Technology, Ireland; Valeo Vision Systems, Ireland; Valeo.ai, France, 2019.

[20] A. Kumar, *Robot Operating System 2 (ROS 2): Introduction and Getting Started*, MakerPro, 2020. [Online]. Available: https://maker.pro/ros/tutorial/robot-operating-system-2-ros-2-introduction-and-getting-started

[21] *Why Gazebo?*, Open Source Robotics Foundation, 2014. [Online]. Available: http://gazebosim.org/

[22] *SDFormat*, Open Source Robotics Foundation, 2020. [Online]. Available: http://sdformat.org/

[23] *Turtlebot3*, Robotis, 2020. [Online]. Available: https://emanual.robotis.com/

[24] A. Choudhury, *Tensorflow vs. Keras: Which one should you choose?*, Robotis, 2019. [Online]. Available: https://analyticsindiamag.com/

[25] *Keras*, Keras, 2020. [Online]. Available: https://keras.io/

[26] *Installing ROS 2 via Debian Packages*, Open Source Robotics Foundation, 2020. [Online]. Available: https://index.ros.org/doc/ros2/Installation/Dashing/Linux-Install-Debians/

[27] *Install Gazebo using Ubuntu packages*, Open Source Robotics Foundation, 2014. [Online]. Available: http://gazebosim.org/tutorials?tut=install_ubuntu